Dependency management for the Eclipse ecosystem

Daniel Le Berre, Pascal Rapicault

Dependency Management for the Eclipse Ecosystem

Eclipse p2, metadata and resolution

Daniel Le Berre*
Univ Lille Nord de France, F-59000 Lille, France
UArtois, CRIL, F-62307 Lens, France
CNRS, UMR 8188, F-62307 Lens, France
leberre@cril.univ-artois.fr

Pascal Rapicault
IBM Rational
Ottawa, Ontario, Canada
pascal_rapicault@ca.ibm.com

ABSTRACT

One of the strength of Eclipse, the well-known open platform for software development, is its extensibility made possible by the built-in pluggability mechanisms. However those pluggability mechanisms only reveal their full potential when extensions created by others are made easy to distribute and obtain. The purpose of Eclipse p2 project is to build a platform addressing the challenges of distribution and obtention of Eclipse and its extensions, which poses the same dependency management issues than for component based systems. This paper focuses on the dependency management aspect of p2. It describes the metadata used to express dependencies, the overall functioning of our resolver and a description of our propositional constraints based encoding. To conclude we describe the challenges to address in future releases.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Extensibility; D.2.13 [**Reusable Software**]: Reusable libraries; F.4.1 [**Mathematical Logic**]: Logic and constraint programming

General Terms

Algorithms, Design, Experimentation

Keywords

OSGi, dependency management, boolean encoding

1. INTRODUCTION

Eclipse is a very popular open platform mainly written in Java and designed from the ground up as an integration platform for software development tools but also for rich client applications [13]. As the Eclipse ecosystem becomes more and more important, the Eclipse platform itself and the vertical platforms built on Eclipse all rely on the concept of extensibility, and as such the necessity of a mechanism to acquire those extensions is primordial. To that end, almost since its inception, Eclipse featured an extension acquisition mechanism named Update Manager. However over time, as inter plug-in dependencies became more complex and expressed at a finer grain, and more versions of each component was made available, limitations were being discovered in Update Manager which were hindering the adoption and retention of Eclipse. The term "plug-in hell" was coined. It is at that time that we started to work on Eclipse p2 with the goal of building a "right-grained" provisioning platform attempting to address the challenges that Update Manager had been faced with.

One of the first challenge was heterogeneity in the set of things being deployed since over time it had become clear that most OSGi- and Eclipse-based applications needed to have a manageable way to interact with their environment (e.g JRE, Windows registry keys, etc.).

Second was the need to address in one platform the diversity of provisioning scenarios and offer a solution that would work against controlled repositories -similar to the case of linux packages managed by a specific Linux distribution- or uncontrolled repositories, would allow for fully automated solutions or user-driven ones, or would sport the delivery of extensions as well as complete products.

Finally, and the most important challenge, was to solve the "plug-in hell" that was partially rooted in the non modular way of acquiring components since Update Manager forced extensions to be installed by a special abstraction one level above the actual extension itself. The term "right-grained" provisioning is a response to this problem and indicates that p2 is not an obstruction to the granularity of what a user would want to make available or obtain.

In order to achieve this goal of "right-grained" provisioning, the efficiency, reliability and scalability of the dependency resolver was key. Having learnt from our experience of authoring the OSGi runtime resolver for Equinox, it was obvious that we would need to base our dependency analysis mechanism on proven solver techniques. Coincidentally, later that year, the work of OPIUM[16] and EDOS[12] backed up our intuition on the usability and maturity of a SAT-based approach to address the problem. The dependency problem for Eclipse is closer to the problem addressed by the follow-up to EDOS project, the Mancoosi Project[1, 15], that is the problem of updating complex open source en-

*The work on explanations detailed in section 4.4 has been supported in part by Genuitec company.

IWOCE'09, August 24, 2009, Amsterdam, The Netherlands.

vironments. Nevertheless, dependency constraints[3] studied in both cases are significantly different. In this paper we are presenting the p2 metadata and the motivation for some of these constructs, detail the implementation of our resolver and conclude by the challenges we are interested in addressing in future releases.

2. P2 METADATA

Core to the majority of installers that deal with composable systems (e.g RPM, Debian, etc.) lies a concept of metadata. One of the goals of this metadata, and the point of focus of this paper, is to capture the dependencies that exist between the components of the system and thus to find missing dependencies or to validate dependencies of a system before it is being modified. As described previously, p2 is intended to deal with more than just the typical Eclipse constructs of OSGi bundles. As such, despite the presence of dependency information in the OSGi bundles composing most of Eclipse applications, p2 abstracts dependencies from the elements being delivered in an entity called an *Installable Unit* (also referred to as *IU*). We now introduce the three kinds of installable units that p2 defines.

2.1 Anatomy of an installable unit

An installable unit, the simplest construct, has the following attributes:

An identifier A string naming the installable unit.

A version The version of the installable unit. The combination identifier and version is treated like a unique ID. We will refer to versions of an installable unit to mean a set of installable unit sharing the same identifier but a different version attribute.

A set of capabilities A capability is the way for the installable unit to expose to the rest of the world what it has to offer. This is just a namespace, a name and a version. Namespace and name are strings. The namespace is here to prevent name collision and avoid having everyone adhere to name mangling conventions.

A set of requirements A conjunction of requirements. A requirement is the way for the IU to express its needs. Requirements are satisfied by capabilities. A requirement is composed of a namespace, a name and a version range [1]. In addition to these usual concepts, a requirement can have a filter (under the form of an LDAP filter [8]) which allows for its enablement or disablement depending on the environment where the IU will be installed, and it can also be marked optional meaning that failing to satisfy the requirement does not prevent the IU from being installable. Finally there is a concept of greed discussed later in this section.

An enablement filter An enablement filter indicates in which contexts an installable unit can be installed. Here again the filter will pass or fail depending on the environment in which the IU will be installed.

[1]A version range is expressed by two version number separated by a comma, and surrounded by an angle bracket, meaning value included, or a parenthesis, meaning value excluded.

Greed	Optional	Semantics
true	false	this is a "strong" requirement.
true	true	this is a "weak" requirement.
false	true	this is a "weakest" requirement, where the match will not be brought in.
false	false	this indicates a case where the requirement has to be satisfied but the IU with this requirement wants this to be brought in by another one. Such a need will be presented in 2.3.

Table 1: Greed and optional interaction.

A singleton flag This flag, when set to true, will prevent a system to contain another version of the installable unit with the same identifier.

An update descriptor The identifier and a version range identifying predecessors to this IU. Making this relationship explicit allows to deal with IUs being renamed or avoid undesirable update paths.

An example of an Installable unit representing the SWT bundle is given in Figure 1. The few things to notice are the usage of namespace to avoid clashes between the Java packages and the IU identifier; the usage of singleton because no two versions of this bundle can be installed in the same eclipse instance; the "typing" of the IU as being a bundle (see namespace `org.eclipse.equinox.p2.type` valued to `bundle`); and the identification of the IU by providing a capability in the `org.eclipse.equinox.p2.iu` namespace.

Now, let's come back on requirements and detail the semantics of greed and optional. By default, a requirement is "strong"[2] (optional is false, greed is true). This means that the IU can only be installed if the requirement is met. If a "strong" requirement is guarded by a filter that does not pass, the requirement is ignored. When the optional flag is set to true, then a requirement becomes "weak" and it does not have to be satisfied for the IU to be installed. However any IU potentially satisfying this requirement will be considered, and a best effort will be made to satisfy the requirement.

When it comes to greed, this is a rather atypical concept that we have added to control the addition of IUs as part of the potential IUs to install in order to satisfy the user request. When the greed is true (default case, and the case for strong requirements), the IU satisfying the dependencies are added to the pool of potential candidates. However when the greed is set to false, such a requirement relies on other dependencies from its own IU or others to bring in what is necessary for its satisfaction. This is used in Figure 1 to capture the fact that even though we have an optional dependency on `org.mozilla.xpcom` we don't want to try to satisfy it eagerly. As such, this optional and non greedy requirement is weaker than a typical optional dependency. Table 1 reviews the four combinations of greed and optionality.

[2]Strong is weaker in our context than the notion of strong dependency introduced in [3]

```
id=org.eclipse.swt, version=3.5.0, singleton=true
Capabilities:
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.swt, version=3.5.0}
    {namespace=org.eclipse.equinox.p2.eclipse.type name=bundle version=1.0.0}
    {namespace=java.package, name=org.eclipse.swt.graphics, version=1.0.0}
    {namespace=java.package, name=org.eclipse.swt.layout, version=1.2.0}
Requirements:
    {namespace=java.package, name=org.eclipse.swt.accessibility2, range=[1.0.0,2.0.0), optional=true, filter=(&(os=linux))}
    {namespace=java.package, name=org.mozilla.xpcom, range=[1.0.0, 1.1.0), optional=true, greed=false}
Updates:
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.swt, range=[0.0.0, 3.5.0)}
```

Figure 1: An IU representing the SWT bundle

2.2 Installable unit patch

2.2.1 The need for patches

So far, the concept of IU is pretty much on par with what most package managers are offering. However what is interesting is the different usage we have observed of this metadata and the implication it has on the rest of the system. Indeed, most people building on top of Eclipse are delivering "products" or "subsystems" and as such they want to guarantee that their customer is getting what has been tested. Failing to do this could result in a non stable product, maintenance nightmare and unsatisfied customers. However in an ecosystem where products can be mixed and where repositories can not be used as control points [3], guaranteeing a functional system is harder. Consequently, to palliate to these possible problems, product producers are using installable units as a grouping mechanism (also referred to as *group*) serving three goals:

1. Facilitate the reusability of a set of functionality by aggregating under one group a set of installable units.

2. Capture a particular configuration of the system, and thus group under one IU an extensible element and a default implementation.

3. Lock down the dependencies on installable units being used, which limits the variability of what can be installable and thus guarantees reproducibility of an installation independently of the content of the repository.

These three goals are visible in the abridged version of the Platform group shown in Figure 2. It shows the grouping of a set of unrelated functions together to facilitate reuse; how the default setup of the help system is being delivered using the `org.eclipse.help.jetty` IU[4]; and finally the ranges expressed all show the desire to precisely control a particular version of each IU being delivered.

The counter part of the lock down which is used extensively throughout Eclipse, is that it makes the delivery of service (e.g. the replacement of a particular IU by another one) complex for the following reasons:

1. Products are often made of groups, themselves recursively composed of other groups (in the Figure 2, the Platform group includes the RCP group), which can make for a rather vast ripple on effect all throughout the system when a low level component needs to be serviced.

2. Not all groups deployed on the user's machine are in the control of the same organization. For example, someone can be running a composition of IBM and Artois University products (both including the Eclipse Platform group), but the Platform group is controlled by the Eclipse open source community. Therefore when the Platform team needs to deliver a fix to a user, it simply can not require all the referring groups to be updated.

3. Not all the dependencies onto a particular IU are known ahead of time.

2.2.2 Anatomy of an Installation Unit Patch

To palliate to these problems, p2 provides the concept of *installable unit patch* (referred to as *patch*). The simplest way to think of a patch is as a mechanism that can reach into any IU and change any requirement. A patch is a regular installable unit to which three concepts have been added:

Requirement change It represents the changes made to installable units. It is a set of requirement pairs where each pair represents a rewriting rule. In the rewriting rule, the left part captures the original requirement that needs to be replaced and the right part captures the resulting requirement. No constraints are applied to the capturing requirement or the new value, the replacement could widen the range, or narrow it, or completely change the requirement (e.g. remove a filter, change optionality, etc.). In order to provide flexibility, the capturing requirement applies on an original requirement if the ranges expressed on the two requirements intersect.

Applicability scope It identifies the installable units to which the patch should be applied. It has enough flexibility to patch all the IUs of the system where a requirement change is applicable, or target just a few IUs. The IUs to patch are identified by requirements.

Lifecycle It indicates when the installable unit patch can be applied. This can be seen as a precondition that has to be satisfied for the patch to be applied. It represents the "when" to apply the patch whereas the applicability scope represents the "what" to patch. It takes the form of a non greedy requirement.

[3]Controlled repository is the approach taken by a majority of linux distributions.

[4]The Eclipse help system allows different web-servers to be used.

```
id=org.eclipse.platform.feature.group, version=3.5.0.v2009
Capabilities:
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.platform.feature.group, version=3.5.0.v2009}
Requirements:
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.rcp.feature.group, range=[3.1.0.v2009,3.1.0.v2009]}
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.ant.core, range=[3.2.0.v2009,3.2.0.v2009]}
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.ant.ui, range=[1.0.0.v2008,1.0.0.v2008]}
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.help, range=[4.0.0.v2009,4.0.0.v2009]}
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.help.jetty, range=[4.0.0.v2009,4.0.0.v2009]}
Updates:
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.platform.feature.group, range=(3.4.0, 3.5.0.v2009]}
```

Figure 2: An IU representing the Plaform subsystem of Eclipse

The example in Figure 3 shows a patch replacing the IUs `org.eclipse.ant.core` and `org.eclipse.ant.ui`. This patch will only try to replace the references to these two IUs from the Platform group (see applicability scope) and will only be applied if the `my.product` IU is installed.

We have shown how installable unit patches avoid a large ripple-on change across metadata by "rewriting" requirements and also how updates to an IU could be delivered without knowing all the dependencies on it. Given the alteration possibilities, another possible use of patches is to add or remove dependencies in an installable unit you would want to reuse but had unsatisfactory dependencies.

2.3 Installable unit fragment

The third and last concept of p2 metadata is the *installable unit fragment*. This concept inspired from the OSGi fragment concept [2] aims at providing a mechanism to augment an installable unit by adding to it properties or configuration information [5]. The need for this stems from the desire to make installable units as reusable as possible and thus to extract out of an installable unit the configuration information that would otherwise bind it to a particular environment. The canonical example of this is how p2 chose to handle the delivery of OSGi start levels. Some of the OSGi bundles delivered by p2 need to be configured with an information indicating when they can be started. Because the IU for OSGi bundles can be reused in different applications with different configuration needs, they can not carry the start level information so that information is separated out into an installable unit fragment. This fragment is then attached to the IU(s) it is augmenting (referred to as *host*) at resolution time and treated as part of the host during the installation. Fragments also offer the ability to attach to several IUs and thus deliver default configuration information to all of them, thus reducing the configuration burden.

An installable unit fragment augments the concept of installable unit by adding the concept of host requirement. This identifies the host or hosts to which the fragment applies and is expressed by a set of requirements. These requirements have to be satisfied for the fragment to be installable.

What is interesting in the expression of host requirement is the importance of the concept of greed, especially when dealing with a fragment delivering default configuration information. Indeed, in those cases, even though you have a fragment that can apply to multiple IUs does not mean that you want to install all the IUs to which it applies. To indicate this, the host requirement is set to be non greedy. This is the case described in the fourth line of Table 1. It is used in Eclipse by the IU fragment responsible for the delivery of the default start level to every OSGi bundles installed in the system (see Figure 4). To perform its job, the host requirement of this fragment requires "IUs that are bundles" (`namespace=org.eclipse.equinox.p2.eclipse.type, name=bundle, range=[1.0.0], greed=false`) which matches the capabilities of the same namespace and name provided by IUs like the SWT one (see Figure 1). If the greed attribute had been set to true, then the resolver would have brought in all the IUs delivering bundles thus growing unnecessarily the set of IUs that would have been added to the pool of potential IUs for the solution, whereas what is intended is to have this IU only attach to the bundles that will be brought through other dependencies. To be pedantic, in the case of the current Eclipse release, the size of the problem would have been multiplied by a factor 10 since the Eclipse SDK is made of about 380 IUs and the Galileo repository [6] contains about 3800 IUs.

3. RESOLVER OVERVIEW

This section describes the overall functioning of our resolver. However the discussion on the propositional constraints encoding and explanation support appears in the next section.

Before detailing the overall solver, it is worth mentioning how p2 manages the installed software. p2 has a concept of profile which keeps track of two key information: the list of all the Installable Units installed, and the set of *root installable units*. The root IUs are not a new kind of installable units, they are installable units that are remembered as having been explicitly asked for installation. These roots are essential for installation, uninstallation and update, since they are used as strict constraints that can't be violated, thus for example avoiding the uninstallation of an IU when installing another one.

3.1 Resolution

p2 resolution process is logically organized in 5 phases:

Change request processing Given a *change request* capturing the desire to install or uninstall an installable unit, a future root set representing the application of this request over the initial root set is produced.

Slicing For each element in the future root set, the slicing produces a transitive closure of all the IUs (referred as

[5]These are not detailed in this paper because their content and representation is irrelevant to the discussion

[6]http://download.eclipse.org/releases/galileo

```
id=org.eclipse.ant.critical.fix,version=1.0.0
Capabilities:
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.ant.core.critical.fix, version=3.5.0.v2009}
Requirement Changes:
    { from={namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.ant.core, range=[3.1.0, 3.4.0)},
      to={namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.ant.core, range=[3.4.3,3.4.3]}}
    { from={namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.ant.ui, range=[1.0.0.v2008,1.0.0.v2008]},
      to={namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.ant.ui, range=[1.1.0.v2009,1.1.0.v2009]}}
Applicability Scope:
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.platform.feature.group, range=[3.5.0.v2009,3.5.0.v2009]}
Lifecycle:
    {namespace=org.eclipse.equinox.p2.iu, name=my.product, range=[1.0.0,1.0.0], greed=false}
Updates:
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.platform.feature.group, range=[0.0.0, 3.5.0.v2009)}
```

Figure 3: Example of installable unit patch, patching the Platform group.

```
id=osgi.bundle.default.startlevel, version=1.0.0
Hosts:
    { {namespace=org.eclipse.equinox.p2.eclipse.type, name=bundle, range=[1.0.0], greed=false} }
Capabilities:
    {namespace=org.eclipse.equinox.p2.iu, name=osgi.bundle.default.startlevel, version=1.0.0}
```

Figure 4: Installable unit fragment attaching to any bundle.

slice) that could potentially be part of the final solution of the resolution process by consulting all repositories also passed in. This transitive closure is done with only taking into account enough context[7] to evaluate the various filters but without worrying if any IU being added could be colliding with any others. For each IU, it looks at each requirement, queries the repositories for matches, and add the results of the query to a set of IU to process. In the case of fragment IUs, the host requirement is also treated and for patches the replacement value contained in the requirement changes are as well. If the requirement greed is set to false, or if the requirement enablement filter is not evaluating to true, the process is short-circuited and the next requirement is considered. The slicing phase reduces the dependency problem to the only IUs applicable to a given setting (the OS for instance). Another solution would have been to encode everything into constraints. Our approach here is to use the constraints solver only for non trivial matters.

Projection/encoding The goal of the projection phase is to transform all the installable units of the slice and their dependencies into a system of propositional constraints (see section 4 for details). For each IU, each requirement [8] is queried against the slice and each resulting IU is encoded. In contrast to the slicing phase, the greed flag is ignored and every requirement, except when filtered, is processed since at that point the slicing has isolated the elements that will be part of the solution. It is also during this phase that patches are applied. For this, for each IU being processed, applicable patches (matching the applicability scope) are searched for in the slice and applied. Since patches may or may not be part of the final solution but logically "modifies" the original requirements, special care has to be taken to have a variable representing the initial IU with the patch applied on it, and another one with the patch not applied on it. This is detailed in section 4.3. In addition to this transformation, this phase also fills in a data structure keeping track of every potential hosts for each fragment IU. Finally for each root the corresponding variable in the SAT encoding is set to true to capture the intent of the solution being searched. The generation of the propositional constraints completed, the optimization function is being generated.

PBO-solving The result of the projection is passed to the pseudo boolean solver SAT4J[10] which is responsible for finding an assignment.

Solution extraction From the assignment returned by the solver, a solution is extracted and the final attachment of IU fragment to their host(s) is concretized. In case of failure the solver is invoked to produce an explanation (see section 4.4).

3.2 Updating

In p2, the detection and installation of all applicable updates is not a completely automated process and no simple function in the p2 API is provided to perform this operation. Instead, the only mechanism provided is a function which given a root IU and a set of repositories returns a set of candidates IUs that are updates of the given IU[9]. This allows for selectively searching for updates or looking for an update of everything that is installed. Once all the candidates have been gathered, a subset of the matches are picked and a change request is created and passed to the resolver for validation. Because p2 is a platform, the picking responsibility is left to code outside of the scope of the core of p2. For example, the graphical interface available in the Eclipse

[7]The context can be seen as a map of key/value pair

[8]Here again, requirement host of IU fragments, replacement values in requirement changes of patches as well as applicability scope are treated.

[9]Each candidate has its update descriptor match the IU it is an update of.

SDK performs filtering and the user is only presented with the most recent version of each IU to select from. The user input obtained, a corresponding profile change request is created and passed to the resolver for validation.

The obvious drawback of this approach is that the user may have to go through several iterations of selection to find a suitable alignment of IUs to update to. However, even if p2 could provide some logic to return the most pertinent and up-to-date set of IUs using techniques like MAX-SAT [4], the results would only be relevant in case of a completely automated update mechanism, since giving the opportunity to the user to pick something from the set of updates available could invalidate the solution initially returned and thus potentially lead to similar problems.

4. CONSTRAINTS ENCODING

In the following, we describe the encoding of p2 installation problem into propositional constraints, i.e. clauses or cardinality constraints. We also provide some examples of problems generated with that encoding.

$prov(IU)$ denotes the set of capabilities provided by the installable unit IU and $req(IU)$ denotes the set of capabilities required by the installable unit IU. $alt(cap) = \{IU_k | cap \in prov(IU_k)\}$ denotes the set of IUs providing a given capability. Finally, $optReq(IU)$ denotes the optional requirements of a given IU, and $versions(IU_x)$ denote the ordered set of IUs sharing the same identifier as IU_x but having different version attribute ($IU_x \in versions(IU_x)$), from the latest to the oldest.

4.1 Basic encoding

Each requirement of the form "IU_i requires capability cap_j" is represented by a simple binary (Horn) clause

$$IU_i \rightarrow cap_j$$

So, for each IU_i the requirements are expressed by a conjunction of binary clauses

$$\bigwedge_{cap_j \in req(IU_i)} IU_i \rightarrow cap_j$$

The alternatives for a given capability is given by the clause

$$cap_i \rightarrow IU_{j_1} \vee IU_{j_2} \vee ... \vee IU_{j_n}$$

where $IU_{j_x} \in alt(cap_i)$.

Since we are only interested in the IUs to install, the above two constraints can be aggregated into a conjunction of constraints:

$$f(IU_i) = \bigwedge_{cap_j \in req(IU_i),} (IU_i \rightarrow \bigvee_{IU_x \in alt(cap_j)} IU_x) \quad (1)$$

Note that there is the specific case of $alt(cap_j) = \emptyset$ which means that IU_i cannot be installed due to missing requirements. In that case, the unit clause $\neg IU_i$ is generated.

Some installation units cannot be installed together (e.g. because of the singleton attribute set to true). This can be modeled either with a conjunction of binary negative clauses

$$\bigwedge_{versions(IU_v)=\{IU_v^1,...,IU_v^n\},1 \leq i < j \leq n} (\neg IU_{v_i} \vee \neg IU_{v_j})$$

or equivalently with a single cardinality constraint:

$$(\sum_{IU_v^x \in versions(IU_v)} IU_v^x) \leq 1 \quad (2)$$

We use the second option because our solver manages those constraints natively and because it makes the explanation support easier to implement (see 4.4 for details).

Finally, the user wants to install the installable units identified by the roots. This is modeled with unit clauses:

$$\bigwedge_{UI_i \in rootIUs} UI_i \quad (3)$$

Summing up, the constraints (1), (2) and (3) altogether form an instance of a classical NP- complete SAT problem. That encoding is basically the encoding presented in Edos[12] and Opium [16] and used more recently in OpenSuse 11[10].

4.2 Encoding of optionality

One of the specificity of p2 is the semantic of "weak" dependencies expressed using the `greed` and `optional` attributes. We do not have to worry here about greedy dependencies since there are simply ignored during the slicing stage. An IU IU_i may have optional dependencies to IU IU_j means that IU_j is not mandatory to use IU_i, so IU_i can be installed successfully if IU_j is not available. However, it is expected that p2 should favor the installation of optional packages if possible, i.e. that all optional packages that could be installed are indeed installed. In Figure 1, one can see that SWT has two optional dependencies on SWT accessibility2 and Mozilla XPCOM. The encoding of optional packages is done by creating two specific propositional variables: Abs_{cap} denotes the fact the capability cap is optional, and $Noop_{IU_i}$ is a variable to be satisfied in case none of the optional capabilities of IU_i can be installed. The first set of constraints expresses how to satisfy the required capabilities:

$$\bigwedge_{cap_j \in optReq(IU_i)} (Abs_{cap_j} \rightarrow \bigvee_{IU_x \in alt(cap_j)} IU_x) \quad (4)$$

The second set of constraints expresses that if $Noop_{IU_i}$ is true then all the abstract capability variables must be false, i.e. that $Noop_{UI_i}$ can only be set to true when none of the optional dependencies could be installed.

$$\bigwedge_{cap_j \in optReq(IU_i)} (Noop_{IU_i} \rightarrow \neg Abs_{cap_j}) \quad (5)$$

Note that such set of constraints could also have been expressed equivalently by a single pseudo boolean constraint:

$$(\sum_{cap_j \in optReq(IU_i)} Abs_{cap_j}) + n \times Noop_{IU_i} \leq n$$

where $n = |optReq(IU_i)|$.

We used the first option in our encoding since it looked easier to understand. The second option could be used in order to reduce the number of constraints used in the solver. Contrariwise to the encoding of the singleton attribute, there is no need to use a single constraint here because optionality

[10] `http://en.opensuse.org/Package_Management/Sat_Solver/Basics`

encoding constraints cannot prevent an IU to be installed, thus cannot be part of an explanation.

Finally, we express that IU_i has optional dependencies using a disjunction ending with the $Noop_{IU_i}$ variable. That way, even if none of the optional requirements can be installed, the constraint can still be satisfied by setting $Noop_{IU_i}$ to true.

$$IU_i \rightarrow \bigvee_{cap_j \in optReq(IU_i)} Abs_{cap_j} \vee Noop_{IU_i} \quad (6)$$

EXAMPLE 1. *Let's see how to encode the optional dependencies of SWT on accessibility2 and xpcom shown in Figure 1:*
$Abs_{accessibility2} \rightarrow IU^{1.0}_{accessibility2}$
$\neg Abs_{accessibility2} \vee \neg Noop_{IU_{SWT}}$
$Abs_{xpcom} \rightarrow IU^{1.1}_{xpcom}$
$\neg Abs_{xpcom} \vee \neg Noop_{SWT}$
$IU_{SWT} \rightarrow Abs_{acessibility2} \vee Abs_{xpcom} \vee Noop_{SWT}$
An alternative encoding would be:
$Abs_{accessibility2} \rightarrow IU^{1.0}_{accessibility2}$
$Abs_{xpcom} \rightarrow IU^{1.1}_{xpcom}$
$Abs_{accessibility2} + Abs_{xpcom} + 2 \times Noop_{IU_{SWT}} \leq 2$
$IU_{SWT} \rightarrow Abs_{acessibility2} \vee Abs_{xpcom} \vee Noop_{SWT}$

That encoding of optionality was the original one that shipped with Eclipse 3.4. The encoding has evolved since then. This will be discussed in section 4.5.

4.3 Encoding of patches

Applying a patch from the encoding point of view only applies to requirements changes (see section 2.2), i.e. it means to enable or disable some dependencies according to the application or not of a given patch. We denote by $patchedReqs(IU, p)$ the set of pairs $< old, new >$ of the installable unit IU denoting the rewriting rules of patch p in the requirements of IUs.

We associate to each patch a new propositional variable. We introduce that variable in dependency constraints (1) and (4) the following way:

- Negatively when the patch introduces a new dependency.
- Positively when the patch removes an existing dependency.

It can be summarize that way:

$$\bigwedge_{<old,new> \in patchedReqs(IU,p)} (p \rightarrow encode(new)) \wedge (encode(old) \vee p)$$

where $encode(x)$ denote the encoding of a regular or an optional dependency. The patch encoding changes only the encoding of the requirements affected by a patch.

EXAMPLE 2. *The patch shown in Figure 3 would lead to the following encoding of the dependencies for the IU Platform:*
$IU_{Platform} \rightarrow IU^{3.1.0}_{ant-core} \vee patch$ *(a)*
$patch \wedge IU_{Platform} \rightarrow IU^{3.4.3}_{ant-core}$ *(b)*
$IU_{Platform} \rightarrow IU^{1.0.0}_{ant-ui} \vee patch$ *(a)*
$patch \wedge IU_{Platform} \rightarrow IU^{1.1.0}_{ant-ui}$ *(b)*
$IU_{Platform} \rightarrow IU^{4.0}_{help}$ *(c)*
$IU_{Platform} \rightarrow IU^{4.0}_{jetty}$ *(c)*
$IU_{Platform} \rightarrow IU^{3.1}_{rcp}$ *(c)*

If the patch is enabled (i.e. the propositional variable patch is set to true) then the initial constraints (a) are disabled while the new dependencies (b) are enabled. If the patch is not enabled (i.e. the propositional variable patch is set to false), then the new dependencies (b) are disabled and the original optional dependencies (a) apply. Note that the requirements untouched by the patch (c) do not see their encoding changed.

4.4 When things go wrong: explanation

Explanation is key to help the user understands why a change request cannot be fulfilled. In the above encoding, one can note that there are only two reasons that could prevent a request to succeed:

- At least one of the required IUs is missing.
- The request requires two IUs sharing the same identifier but with different versions that cannot be installed together due to the singleton attribute on at least one of those IUs (see for instance Figure 5).

As a consequence, it is not hard to check why a request cannot be completed. However users expect the explanation to be returned in terms of IUs they know about, the root IUs and the IUs that they are trying to install, and would be confused if provided with just the low level dependency errors. In practice, it means that knowing why a problem occurred is not sufficient. It is important to be able to detail the whole dependencies from the root to the actual cause of the problem. For instance, in Figure 5, the user cannot install the Eclipse Platform IU version 3.5.0.20090528 because the Eclipse SDK version 3.5.0.20090430 is already installed and both of them rely on different versions of Eclipse RCP that is a singleton.

Let S be the set of the constraints encoding presented in the previous sections. From a logical point of view, it is possible to compute one minimal subset S' of the constraints that cannot be satisfied altogether: $S' \subseteq S, S' \models \bot, \nexists S'' \subset S' | S'' \models \bot$. Such set of constraints is often called a MUS (minimal unsatisfiable subformula). S' is an explanation of the impossibility to fulfill the request. If the subset contains a negated literal (specific case of Equation (1), $\neg UI_x \in S'$) then the global explanation is a missing requirement, i.e. the request cannot be completed because IU_x cannot be found in the user's repositories. If the subset contains a cardinality constraints ($\sum IU^x_v \leq 1 \in S'$), then the global explanation is a singleton attribute violation, i.e. the request cannot be completed because it requires several versions of IU_v. Note that if we decided to use a clausal encoding instead of the cardinality constraints encoding, we would have lost the one to one mapping between the original dependencies and the constraints of our encoding.

There are several ways in practice to compute S' from S. The ones based on local search algorithms[14] detect constraints that are likely to be part of S' among the most falsified ones during the search and compute S' in a second step using a complete SAT solver. A more recent and widely used approach is based on the analysis of the last conflict found by a conflict driven SAT solver[17]. Such approach requires some changes in the SAT solver to keep track of all resolutions steps and does not ensure that the computed subset $S'' \subseteq S$ is minimal. A third approach is to rely on a new encoding of the problem into an optimization problem

using selector variables [11]: it is possible to use an optimization function on selector variables to compute a set S' of minimal size. Finally, a generic approach to explanation in constraints solvers was proposed in [9] and implemented in Ilog solver: QuickXplain. The main advantage of such approach is to be independent of the underlying solver, and to work with any kind of constraints.

Our approach inherits some ideas from all those approaches. We decided to implement the QuickXplain algorithm in our framework because it is non intrusive (does not require any change to the solver) and works perfectly with mixed constraints (clauses and cardinality constraints in our case). We use selector variables in our encoding to allow the QuickXplain algorithm to enable/disable the constraints when computing S'. Finally, the constraints given to the QuickXplain algorithm are ordered in decreasing order of their activity in the spirit of the local search approaches.

More precisely, we translate S into S'' by adding a new selector variable sel_i to each constraint in S: $S'' = \{sel_i \vee s_i | s_i \in S\}$. Let SEL denotes the set of all added selector variables. Instead of looking for an assignment satisfying S, we are looking for an assignment satisfying S'' under the assumption that all variables in SEL are set to false, $S'' \bigwedge_{sel_i \in SEL} \neg sel_i$[11]. If such assignment exists, it is an assignment satisfying S, so we are done. If it is not the case, then we use a tailored version of the QuickXplain algorithm that makes use of the selector variables to enable/disable constraints in order to compute S'.

4.5 From decision to optimisation

When all the constraints can be satisfied, there are usually many possible solutions, that are not of equal quality for the end user. Here are a few remarks regarding the quality of the expected solution:

1. An IU should not be installed if there is no dependency to it.
2. If several versions of the same bundle exist, the latest one should preferably be used.
3. When optional requirements exist, the optional requirements should be satisfied as much as possible.
4. User installed patches should be applied independently of the consequences of its application (i.e. the version of the IUs forced, the number of installable optional dependencies, etc.).

We are now looking for the "best" solution, not just any solution, i.e. we moved from providing a certificate for the answer to a decision problem (NP-complete from a complexity theory point of view) to return the solution of an optimization problem (NP-hard). Furthermore, we need to solve a multi-criteria optimization problem since it is likely that several IUs do have optional requirements and that several IUs are available in multiple versions.

To solve our problem, we build a linear optimization function to minimize in which the propositional variables are either given a penalty (positive integer) or a reward (negative integer) to prevent or favor their appearance in the computed assignment.

[11]Assumption based satisfiability testing is available in all Minisat[7] inspired solvers (including SAT4J).

- Each version of an IU gets a penalty as a power of 2 proportional to its age, the older it is the more penalized it is:

$$\sum_{IU_v^i \in versions(IU_v)} 2^i \times IU_v^i \quad (7)$$

  That way, each installation of an IU raises at least a penalty equals to one, thus expressing that only required IUs should be installed.

- Each $Noop_x$ variable gets a penalty to favor the installation of optional IU.

$$\sum 2^K \times Noop_x \quad (8)$$

  where K is a constant such that 2^K is greater than the maximum penalty for a version (i.e. $K > max(|versions(IU_v)|)$.

- Each Abs_x variable gets a reward to favor the installation of optional dependencies

$$\sum -2^{K+1} \times Abs_x \quad (9)$$

- Each *patch* variable gets a reward of $n \times -2^{K+3}$ if it is applicable (where n denotes the number of applicable patches), else a penalty of 2^{K+2}

$$\sum_{p_i \in applicablePatches()} n \times -2^{K+3} p_i + \sum_{p_i \notin applicablePatches()} 2^{K+2} p_i \quad (10)$$

The objective function of our optimization problem is thus to minimize (7) + (8) + (9) + (10).

The weights in (7) are not satisfactory since they do not provide a total order on the final solution. Suppose that we have two IUs IU_a and IU_b that are available in respectively 3 and 2 versions (namely IU_a^3, IU_a^2, IU_a^1 and IU_b^2, IU_b^1). The objective function for those IUs is thus

$$IU_a^3 + 2 \times IU_a^2 + 4 \times IU_a^1 + IU_b^2 + 2 \times IU_b^1$$

The best solution for such objective function if both IU_a and IU_b must be installed is obviously to install IU_a^3 and IU_b^2. However, if those two IUs cannot be installed together, the solver will answer that the best option is either to install IU_a^3 and IU_b^1 or IU_a^2 and IU_b^2.

The common approach to solve this problem is to rank each IUs in a total order, $IU_1 < IU_2 < ... < IU_m$, meaning that IU_i is more important than IU_j iff $IU_j < IU_i$. Then the coefficients of the optimization function should be generated in such a way that the sum of the coefficients of IU_j should be smaller than the smallest coefficient of IU_i. In our example, it would mean for instance to use the following optimization function:

$$IU_a^3 + 2 \times IU_a^2 + 4 \times IU_a^1 + 8 \times IU_b^2 + 16 \times IU_b^1$$

In that case, the best option is still to install IU_a^3 and IU_b^2, but the second best option is to install IU_a^2 and IU_b^2.

Unfortunately, as noted before, we are in the context of uncontrolled repositories, so there is no obvious/easy way to order the IUs in a total order, so it was decided to keep the initial solution (7) instead of ranking arbitrarily the IUs.

Another drawback of our objective function appeared recently when new IUs got added to the release repository. The part (9) of the optimization function has the undesirable effect to favor the installation of optional requirements

```
Cannot complete the install because of a conflicting dependency.
  Software being installed: org.eclipse.platform.sdk 3.5.0.I20090528
  Software currently installed: org.eclipse.sdk.ide 3.5.0.I20090430
  Only one of the following can be installed at once:
    org.eclipse.rcp.configuration_root.gtk.linux.x86 1.0.0.I20090430
    org.eclipse.rcp.configuration_root.gtk.linux.x86 1.0.0.I20090528
  Cannot satisfy dependency:
    From: org.eclipse.platform.sdk 3.5.0.I20090528-2000
    To: org.eclipse.rcp.configuration.feature.group [1.0.0.I20090528]
  Cannot satisfy dependency:
    From: org.eclipse.rcp.configuration.feature.group 1.0.0.I20090430
    To: org.eclipse.rcp.configuration_root.gtk.linux.x86 [1.0.0.I20090430]
  Cannot satisfy dependency:
    From: org.eclipse.rcp.configuration.feature.group 1.0.0.I20090528
    To: org.eclipse.rcp.configuration_root.gtk.linux.x86 [1.0.0.I20090528]
  Cannot satisfy dependency:
    From: org.eclipse.sdk.ide 3.5.0.I20090430-2300
    To: org.eclipse.rcp.configuration.feature.group [1.0.0.I20090430]
```

Figure 5: Example of explanation

of an IU even if such IU is not installed. A contextualization of the reward is necessary to fix such problem:

In the constraints our first solution to this problem has been to contextualize (4):

$$\bigwedge_{cap_j \in optReq(IU_i)} (Abs_{cap_j} \wedge IU_i \rightarrow \bigvee_{IU_x \in alt(cap_j)} IU_x) \tag{11}$$

That way, the variable Abs_{IU_i} can be forced to true by the objective function but it will not fire the installation of the dependencies if the IU_i is not to be installed. However, that solution has an unexpected behavior in the following test case: suppose that IU_a has an optional dependency on capability b provided by IU_b that has in turn an optional dependency on the capability c provided by IU_c for which we have $alt(IU_c) = \emptyset$. We would generate the following constraints:

$Abs_c \wedge IU_b \rightarrow \bot \equiv \neg Abs_c \vee \neg IU_b$ (a)
$Abs_b \wedge IU_a \rightarrow IU_b$
$IU_a \rightarrow IU_b \vee Noop_{IU_a}$

Since we would like to satisfy as many Abs_x variables as possible, the solver would force Abs_c and Abs_b to be set to *true*. As a consequence, IU_b has to be set to false to satisfy (a). This is possible because the dependency to IU_b is optional, else a strong dependency would force IU_b to be set to *true*. Such unexpected behavior is a direct consequence of changing the constraints because of a wrong behavior of the objective function. The fix should not change the constraints but the objective function itself.

In the objective function The other option is to use a non linear optimization function in our problem, i.e. to make the reward a function of both the abstract variable Abs_{cap_i} and IU_j where $cap_i \in optReq(IU_j)$:

$$\sum -2^{K+1} \times Abs_{cap_i} \times IU_j \tag{12}$$

The problem is that our solver does not propose yet an easy way to work with non linear optimization functions. A solution based on the introduction of new variables $y_k \leftrightarrow Abs_{cap_i} \times IU_j$ where y_k replaces $Abs_{cap_i} \times IU_j$ in the objective function and with the additional constraints $y_k \rightarrow Abs_{cap_i} \vee IU_j$,$Abs_{cap_i} \rightarrow y_k$,$IU_j \rightarrow y_k$ should fix that issue.

That last issue was discovered late in the release cycle of Eclipse 3.5 Galileo. Considering the fact that meeting the scenario of the issue of the first option was less likely than introducing a new issue by integrating a barely untested implementation of the second option in SAT4J, the former has been adopted for Eclipse 3.5.

5. CONCLUSION AND PERSPECTIVE

We presented Eclipse p2, a "right-grained" provisioning platform aimed at solving the diversity of provisioning requirements in a componentized world. We focused on presenting the three metadata constructs which allow the assembly of complex products from components:

- Installable unit, to capture dependencies among component at their lowest level, but also to capture lock downs and ease reusability by grouping;
- Installable unit patch, to tweak dependencies of installable units that can't be changed;
- Installable unit fragment, to "augment" an installable unit and as such allow for installable units to be as reusable as possible by staying context free from their used one deployed.

From there we presented the overall functioning of our resolver and gave a description of our SAT-based encoding that is resolved by the SAT4J Pseudo Boolean solver.
We can report that this approach that has been live for more than one year has proven to be reliable, efficient and scalable even when faced with repositories containing more than 10000 installable units and solution involving about 3000 installable units. Furthermore, a direct consequence of our work is the integration of the very same technology to manage dependencies in other Java related products, namely the upcoming major release of Maven (Maven3) and the repository manager Nexus.

That said, we are looking to further improve the metadata and the resolver to facilitate composition and to address the challenges encountered so far. For the metadata, the main changes we want to investigate are:

1. The ability for the line-up information that is usually expressed in groups to be extracted out into a new kind of installable units. Once extracted, this information would no longer be directly encoded as propositional constraints but would instead be used to drive the optimization function. We hope that such a change would facilitate the serviceability by avoiding the need to create patches, but the biggest challenge with this feature would be composability of several of these new IUs.

2. The ability to express capabilities and requirements on other domains than just version and version ranges. For example we are thinking about adding scalars to describe the amount of memory available.

3. Add negation and disjunction to improve the expressiveness of the requirements.

On the resolver, the main changes planned surround:

1. Stability of resolution to not cause updates of IUs that are not directly related to the change request being performed. The canonical example is where requirements toward an IU tolerate several possibilities and the installation of something new cause the update of the IU because a newer version is available in the repositories. We hope that the concept of line-up previously described will help there.

2. Resolution over a set of profiles, in order to perform the coordinated management of several applications meant to work together (e.g. the client needs to be in sync with the server).

3. Speed up the explanation process. Depending on the size of the set of constraints, it can take several minutes to provide the answer which is not ideal in an interactive tool. This is caused by the $n \times log(n)$ satisfiability tests used by the explanation algorithm used (QuickXplain) where n is the size of the input of the algorithm in number of constraints. n can be as big as the number of constraints in the encoding in the worst case.The idea is to ask more information to the SAT solver instead of using a purely external approach as currently in order to minimize n in the spirit of [6, 5].

6. REFERENCES

[1] Mancoosi, Managing the Complexity of the Open Source Infrastructure. http://www.mancoosi.org.

[2] OSGi Service Platform. http://www.osgi.org/Specifications.

[3] Pietro Abate, Jaap Boender, Roberto Di Cosmo, and Stefano Zacchiroli. Strong dependencies between software components. Technical Report 2, Mancoosi - Seventh Framework Programme, May 2009.

[4] Josep Argelich, Ines Lynce, and Joao Marques-Silva. On solving boolean multilevel optimization problems. In *Twenty-First International Joint Conferences on Artificial Intelligence (IJCAI)*, page to appear, Pasadena, California, USA, 2009.

[5] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Efficient generation of unsatisfiability proofs and cores in sat. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2008.

[6] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. A simple and flexible way of computing small unsatisfiable cores in sat modulo theories. In Joao Marques-Silva and Karem A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 334–339. Springer, 2007.

[7] Niklas Eén Niklas Sörensson. An extensible sat-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing, LNCS 2919*, pages 502–518, 2003.

[8] IETF. http://www.ietf.org/rfc/rfc2254.txt.

[9] Ulrich Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In Deborah L. McGuinness and George Ferguson, editors, *AAAI*, pages 167–172. AAAI Press / The MIT Press, 2004.

[10] Daniel Le Berre and Anne Parrain. SAT4J, a SATisfiability library for java. http://www.sat4j.org.

[11] Ines Lynce and Joao P. Marques Silva. On computing minimum unsatisfiable cores. In *SAT*, 2004.

[12] Fabio Mancinelli, Jaap Boender, Roberto di Cosmo, Jérôme Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE06)*, pages 199–208, Tokyo, JAPAN, september 2006. IEEE Computer Society Press.

[13] Mark Powell (NASA) Marc Hoffmann, Gilles J. Iachelini (CSC). Eclipse on rails and rockets. http://live.eclipse.org/node/750.

[14] Bertrand MAZURE, Lakhdar SAIS, and Eric GREGOIRE. Detecting logical inconsistencies. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics(AI/Math'96)*, pages 116–121, Fort Lauderdale (FL-USA), jan 1996.

[15] Ralph Treinen and Stefano Zacchiroli. Solving package dependencies : from Edos to Mancoosi. In *DebConf'8*, Argentine, 2008.

[16] Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Opium: Optimal package install/uninstall manager. In *ICSE*, pages 178–188. IEEE Computer Society, 2007.

[17] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT03)*, 2003.